SOME MODIFIED ALGORITHMS FOR DIJKSTRA'S LONGEST UPSEQUENCE PROBLEM

BY

ROBERT B.K. DEWAR

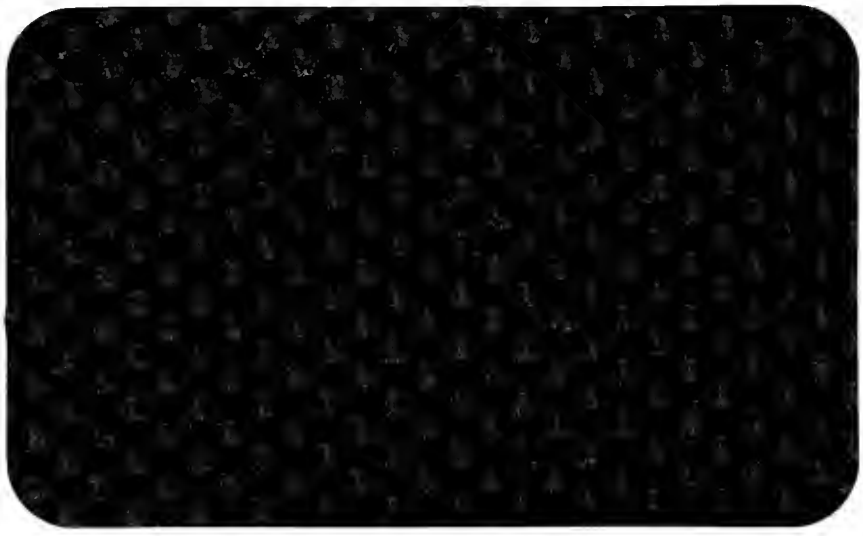
SUSAN M. MERRITT

and

MICHA SHARIR

October 1980

REPORT NO. 026



SOME MODIFIED ALGORITHMS FOR DIJKSTRA'S LONGEST UPSEQUENCE PROBLEM

BY

ROBERT B.K. DEWAR

SUSAN M. MERRITT

and

MICHA SHARIR

October 1980

REPORT NO. 026

This material is based upon work supported by the National Science Foundation under Grant No. MCS-8004349.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Some Modified Algorithms for Dijkstra's Longest Upsequence Problem

Robert B.K. Dewar, Susan M. Merritt and Micha Sharir

Computer Science Department
Courant Institute
New York University

October 1980

ABSTRACT: Two modified merge algorithms for the longest upsequence problem are presented. The algorithms take advantage of natural runs in the input sequence and have a worst case O(nlogn) time complexity when appropriate merging techniques are used, but can be linear if long runs are present in the sequence. The first algorithm is logically equivalent to Dijkstra's algorithm; the second algorithm is based on the first one but uses a different merging technique. Concluding remarks describe how these results evolved out of our work in programming by transformation.

Longest upsequence problem

Given a sequence of n integers, an upsequence is a subsequence which is ordered in nondecreasing order. A subsequence is any subset of the original sequence in which the original order is retained (hence there are 2**n possible subsequences.)

The problem we consider is: Give an algorithm which, given a sequence, will return the length of its longest upsequence. (Note that there may be more than one longest upsequence of the same length.)

Dijkstra's algorithm

Dijkstra's algorithm [Di] examines elements in the original sequence from left to right. When a 'next' element is examined it is inserted into what Dijkstra denotes as the m array (consisting of minimum rightmost elements of upsequences of length 1,2,...). (Here we will refer to the 'm array' as the 'target sequence'.) New sequence elements x are inserted into the target sequence either by being added to the right end (if x is larger than the rightmost element of the partial target sequence) or by 'bumping' an element already in the target sequence. If x = A[n] is the next element of the input sequence,

the heart of the Dijkstra algorithm is:

```
if m[k] <= A[n] then k := k+1; m[k] := A[n];
elseif A[n] < m[1] then m[1] := A[n];
else "establish j such that
          m[j-1] <= A[n] < m[j]";
     m[j] := A[n];
end if;
```

Dikstra uses a binary search technique to determine j, which makes the complexity of his algorithm O(nlogn).

First Modified Merge Algorithm

The algorithm presented here, although logically equivalent to Dijkstra's algorithm, uses a fresh and more efficient approach. Rather than examining the input sequence element by element in a left to right manner, the algorithm proceeds as follows:

1) Divide the input sequence into natural ascending runs r1,r2, ..,rm;

2) Merge runs in a left to right manner using the modified merge operation detailed below (The order of merges is as follows: merge r1 with r2; merge the resulting run with r3; continue merging in this manner r4,..,rm with the leftmost run.)

3) output the length of the final merged sequence, which will be equal to the length of the longest upsequence.

Our nonstandard merge of two adjacent runs operates in a manner rather similar to standard merging. However, whenever the next member p of the right run is smaller than the next member q of the left run, p bumps q, i.e. p is copied into the merged run but q is not, and we advance one step in both runs to continue the merging.

This procedure maintains the following invariant proparty (which shows it to be equivalent to Dijkstra's algorithm): The j-th element from the left in the merged sequence is always the smallest rightmost element of an upsequence of length j in the portion of the original sequence which has entered so far into the merge.

The following program, "longest_upseq", uses the modified merge to solve the longest upsequence problem.

```
program longest_upseq;

loop do
  read(d);
  print(#merge_runs(create_runs(d)));
end loop;

procedure create_runs (d);
  return if exists i in [1..#d-1] | d(i) > d(i+1) then
          [d(1..i)] + create_runs (d(i+1..))
         else [d] end;
end procedure;

procedure merge_runs (rs);
  return if #rs = 1 then rs(1)
         else merge_runs([merge(rs(1),rs(2))] + rs(3..)) end;
end procedure;

procedure merge (a, b);
  return if a = [] then b
         elseif b = [] then a
         elseif a(1) <= b(1) then a(1..1) + merge (a(2..), b)
         else b(1..1) + merge (a(2..), b(2..)) end;
end procedure;

end program;
```

Consider the following example.

Let the input sequence be

1 3 5 7 8 2 9 4 10 6

After identifying the runs we get

1 3 5 7 8 / 2 9 / 4 10 / 6

Merge r1 with r2:

1 2 5 7 8 9 / 4 10 / 6

Merge the new run with r3:

1 2 4 7 8 9 10 / 6

Merge with r4:

1 2 4 6 8 9 10

The length of the longest upsequence is therefore 7.

Complexity of the Algorithm

The merge algorithm presented here can certainly be made O(nlogn). Each merge could be accomplished with k binary insertions where k is the length of the right hand run. In this case the algorithm is at least as good as Dijkstra's. Moreover, merging runs by binary insertion (rather than inserting individual elements) gives better actual performance since the target sequence decreases as successive elements are inserted, and since "tails" of runs can be appended in constant time.

It is of interest to consider a sequence which is made up of a small number of very long natural runs. In this case, while merging done by binary insertion will produce an O(nlogn) algorithm, ordinary tape merging will produce an O(n) algorithm. In fact, a random sequence will have both long runs and short runs. While short runs (say of length 1 or 2) are best merged by binary insertion, long runs are best merged by tape merging.

A hybrid merging algorithm such as the Hwang-Lin binary merging algorithm [Kn] could be used to advantage here. In their analysis Hwang and Lin [HL] show that the maximum number of compares needed to merge m elements with n elements, K(m,n), using their algorithm, is equal to that needed by binary insertion for m=1 or m=2 and is less than that needed by binary insertion for m>2. That is,

[Hwang-Lin] K(m,n) = [binary insertion] K(m,n) = ceil(log(n+1))
for m=1 or m=2

and

[Hwang-Lin] K(m,n) < [binary insertion] K(m,n) for m>2.

Also

[Hwang-Lin] K(m,n) = [tape merge] K(m,n) = m+n-1 for m~n.

Clearly the modified merge algorithm for the longest upsequence problem using the Hwang-Lin binary merge (or some variant such as one of those proposed by Manacher [M1], [M2]) would give slightly better worst case performance than Dijkstra's algorithm. Since the expected time for the Hwang-Lin algorithm would also be better than that for binary insertion, our modified merge would also enjoy better average case performance (however, we do not present a formal analysis here.)

It is also interesting to note that (to our knowledge) a precise lower bound order for the longest upsequence problem is not known. One would either expect future investigations to show that this bound is O(nlogn), or else reveal faster algorithms in which merging techniques might play an important role.

Second Modified Merge Algorithm

Next we consider an interesting variant of the first algorithm. We first observe that we can get a 'mirror-image' of the preceding algorithm by merging runs from right to left, going through each run in descending order (also from right to left) and letting larger elements from the left run bump smaller elements in the right run.

Applied to the same example shown above, this variant will work as follows:

We begin by partitioning the sequence into runs as before:

1 3 5 7 8 / 2 9 / 4 10 / 6

We first merge-to-the-right r3 with r4:

1 3 5 7 8 / 2 9 / 4 10

Then we merge r2 with the new run:

1 3 5 7 8 / 2 9 10

And finally merge r1:

1 3 5 7 8 9 10

Again, the length of the longest upsequence is 7.

In analogy with the invariant maintained by Dijkstra's algorithm and by our first algorithm, the 'reverse' algorithm maintains the following 'reverse' invariant:

Let m denote the target sequence constructed by the reverse algorithm. Let k be the j'th element from the right of m. Then k is the largest leftmost element of an upsequence of length j in the original sequence.

None of this changes our basic approach. But now we propose to mix the two algorithms as follows:

1) Partition the set of all runs into two equal parts, a left one containing the leftmost half of the runs, and a right one, containing the righmost half.

2) Apply the first algorithm to the left part, and the reverse algorithm to the right part, to obtain two final runs.

3) Merge these two runs by using either the first algorithm or its reverse counterpart.

4) The length of the resulting run is then the length of the

longest upsequence.

Considering the same example as before, we again have the following partition into runs:

1 3 5 7 8 / 2 9 / 4 10 / 6

We merge r1 with r2 'to the left', and merge r3 with r4 'to the right' yielding:

1 2 5 7 8 9 / 4 10

Now we merge these runs, say, to the right:

1 2 5 7 8 9 10

and obtain 7 as the length of the longest upsequence.

The correctness of this hybrid algorithm is more difficult to establish than that of the preceding algorithms. It is proved as follows:

Let U1, U2 ... Us be the target sequence obtained by left-merging the left half of the runs, and let V1, V2 ... Vt be the target sequence obtained by right-merging the right half of the runs, but written backwards, so that V1 is its rightmost element, and Vt is the leftmost. Written this way, U is increasing, whereas V is decreasing.

These sequences have the following properties:

(A) For each i in [1..s], Ui is the smallest rightmost member of an upsequence of length i contained in the left half of the original sequence (i.e. in the concatenation A- of the leftmost half of the runs), and there does not exist in A- an upsequence of length > s.

(B) For each j in [1..t], Vj is the largest leftmost member of an upsequence of length j contained in the right half A+ of the original sequence; A+ does not contain an upsequence of length > t.

Now let L be an upsequence contained in the whole of the original sequence A, having p elements in A- and q elements in A+. Write L as

L1, L2 ... Lp, L'q ... L'2, L'1

Then, by properties (A) and (B), we have p <= s, q <= t and

Li >= Ui , for i in [1 .. p],

L'j <= Vj , for j in [1 .. q].

But since L is increasing, we must have

Ui <= Li <= L'j <= Vj , for i and j as above,

and in particular

Up <= Vq.

Consequently, if for some p and q the last inequality fails to hold, there cannot exist an upsequence in A having p elements in A- and q elements in A+. The converse statement is also true. Indeed, suppose that the last inequality does hold. Then, by properties (A) and (B), there exists a p-upsequence in A- ending in Up, and a q-upsequence in A+ starting in Vq. Their concatenation is the desired upsequence.

We have thus proved the following

LEMMA: There exists an upsequence in A having p elements in A- and q elements in A+ if and only if p <= s, q <= t and Up <= Vq.

Let us now suppose that we have left-merged the sequences U and V to obtain a final target sequence W. Note that our bumping merge is such that the length r of W is at least m, and at most m+n. We can thus write r = m + k, where k in [o..n]. Moreover, the k rightmost elements of W (reading from right to left) must be

V1, V2 ... Vk.

Let us now show that A contains an upsequence of length m + k. Write W as

W1, W2 ... Wm, Vk ... V2, V1

Scan the W's from right to left starting at Wm, till a member belonging to the U sequence is found. It is easily seen that elements of U that pass into the merged sequence retain their original positions in the target sequence. Let the first element of U found in W be Ui = Wi. This means that the next element to its right is V(k+m-i). Thus we have Ui <= V(k+m-i), so that by the lemma there exists in A an upsequence of length k+m.

Next we show that there cannot exist a longer upsequence in A. Indeed, suppose that there exists a (k+m+1) upsequence. Since it cannot have more than m elements in A-, it must have at least k+1 elements in A+. Suppose that it has k + h elements in A+. By (B), its leftmost element must be <= V(k+h). However, V(k+1) .. V(k+h) have all bumped some elements of U, so that V(k+h) must have bumped an element of U which is at least h places from the right end of U, and so is <= U(m-h+1). All this implies that

V(k+h) < U(m-h+1)

so that by the lemma such an upsequence cannot exist. This proves our claim.

For the sake of completeness, here is the hybrid algorithm in detail.

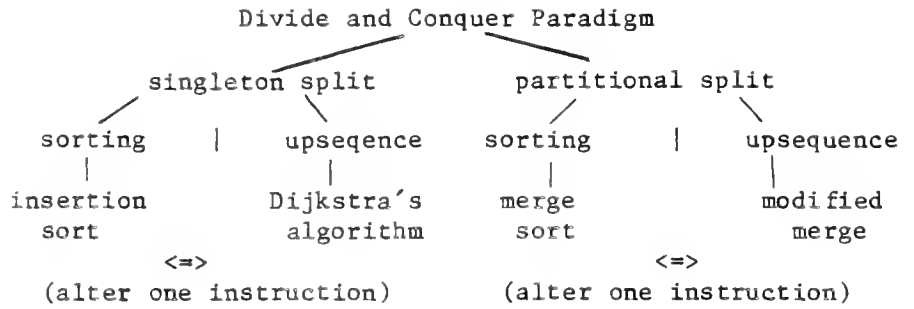
```
program longest_upseq;

loop do
  read(d);
  print(#merge_runs(create_runs(d)));
end loop;

procedure create_runs (d);
  return if exists i in [1..#d-1] | d(i) > d(i+1) then
           [d(1..i)] + create_runs (d(i+1..))
         else [d] end;
end procedure;

procedure merge_runs (rs);
  return
        case #rs of

        (0): [],

        (1): rs(1),

        (2): left_merge(rs(1), rs(2)),

        (3): left_merge(rs(1), right_merge(rs(2),rs(3)))

        else merge_runs([left_merge(rs(1),rs(2))] +
                        rs(3..#rs-2) +
                        [right_merge(rs(#rs-1),rs(#rs))])
        end;
end procedure;

procedure left_merge (a, b);
  return if a = [] then b
         elseif b = [] then a
         elseif a(1) <= b(1) then a(1..1) + left_merge (a(2..), b)
         else b(1..1) + left_merge (a(2..), b(2..)) end;
end procedure;

procedure right_merge(a, b);
  return if b = [] then a
        elseif a = [] then b
        elseif b(#b) >= a(#a) then
                right_merge(a, b(1..#b-1)) + [b(#b)]
        else right_merge(a(1..#a-1), b(1..#b-1)) + [a(#a)] end;
end procedure;

end program;
```

Discussion

Our approach via a modified merge makes the longest upsequence problem a not-so-distant relative of sorting. The striking similarity between the modified merge and the standard merge used in sorting is shown by the fact that in the procedure "merge" shown above, a change from a(2..) to a(1..) in the last line of the if-expression would turn our longest upsequence algorithm into a standard sort-by-merging algorithm. It is possible that further investigation of these similarities would yield other interesting approaches to the longest upsequence problem. Note that Dijkstra's algorithm stands in complete analogy to the binary-insertion sort. (In fact it is the only sequential binary insertion technique which does not require us to move the sorted elements, because of the bumping used, and therefore attains its high efficiency.)

The symmetries between the algorithms for the upsequence problem and the algorithms for sorting are of particular interest to us in the context of our work in transformational programming [DS], [Sh]. In particular we have been investigating the role of the high level specification in programming by transformation [Me]. We have found that it is often the case that more than one high level specification can be given for a particular problem. Moreover, different, well chosen specifications will lead to different derived algorithms. We hypothesize that in some cases the specification style ,or pattern, has implicitly imbedded within it some algorithm synthesis construct or paradigm which determines the structural characteristics of the derived algorithms. One such construct is the "divide and conquer paradigm" discussed by Green and Barstow [GB]. This paradigm refers to the notion of how an algorithm "splits" its input in order to process it. Two alternative methods of splitting the input are the "singleton split" and the "equal-size split" (which we prefer to call the "partitional split".) In sorting, selection and insertion sorts are representative of the singleton method, while quick and merge sorts are representative of the partitional method.

Our merge approaches to the longest upsequence problem grew out of our systematic attempt to apply the partitional method of splitting to this problem. (Note that Dijkstra's algorithm represents the singleton method.) It has been gratifying to discover algorithms which demonstrate the splitting symmmetry within the longest upsequence problem, as well as to discover the analogy between the sorting algorithms and the longest upsequence algorithms which was described above. The following tree summarizes these symmetries.



Finally, it was particularly gratifying to find that the algorithm which emerged for the longest upsequence problem, as a result of our methods, was slightly better than the algorithm previously given by Dijkstra.

R E F E R E N C E S

[Di] Dijkstra, Edsger W. "Some Beautiful Arguments Using Mathemtical Induction," Acta Informatica 13, 1980.

[DS] Dewar, Robert B.K. and Shoenberg, E. "The Elements of SETL Style," Proc. ACM Conf., Detroit, 1979.

[GB] Green, Cordell and Barstow, David R. "On Program Synthesis Knowledge," Artificial Intelligence 10, 1978.

[HL] Hwang, F.K. and Lin, S. "A Simple Algorithm for Merging Two Disjoint Linearly Ordered Sets," SIAM J. Computing, 1,1972.

[Kn] Knuth, Donald E. The Art of Computer Programming, vol. III: Sorting and Searching, Addison-Wesley, 1973.

[M1] Manacher, Glenn K. "Significant Improvements to the Hwang-Lin Merging Algorithm," JACM 26,3, 1979.

[M2] ----------. "The Ford Johnson Sorting Algoithm is Not Optimal," JACM 26,3, 1979.

[Me] Merritt, Susan M. Thesis in progress, Computer Science Dept. New York University, 1980.

[Sh] Sharir. Micha "Some Observations Concerning Formal Differentiation of Set-theoretic Expressions," Computer Science Technical Report 16, CIMS, 1979.

c.1

NYU
Comp. Sci. Dept.
TR-026
AUTHOR
Dewar
TITLE
Some modified algorithms...